



Ahmad Albarasy · Follow

Last edited by Ahmad Albarasy on Dec 19, 2025 

Contributed by



DSaA Lab 6 - Trees

The Islamic University of Gaza
Engineering Faculty
Department of Computer Engineering

Prepared by: Eng. Ahmad Albarasy

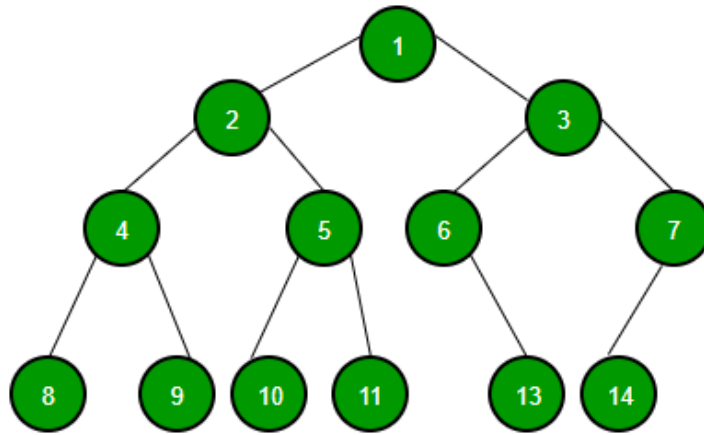
Under the supervision of: Prof. Ahmad Mahdi

In this lab, we are going to explore trees, one of the most important and interesting data structures that are used across various computer science domains to manage data efficiently, primarily due to their ability to represent hierarchical relationships and facilitate fast searching and sorting.

We will introduce the fundamental terminology associated with trees, examine the different types of trees, and study the various tree traversal techniques used to systematically visit their nodes.

Definition & Key Terminology

A tree is a non-linear data structure that stores elements hierarchically. With the exception of the top element, each element in a tree has a parent element and zero or more children elements.



- **Node:** the basic building block of a tree. Each node stores a data element and may have references to one or more child nodes.
- **Root:** The topmost node in a tree which does not have a parent
- **Leaf Node:** A node which does not have any children
- **Internal Node:** A node which has at least one child
- **Sibling Node:** A node is considered a sibling of another node if they share the same parent
- **Ancestors of a Node:** The nodes that lie on the path from the root to a node p , excluding the p itself.
- **Descendants of a Node:** All nodes that are reachable from a node p by repeatedly following child links, excluding p itself.
- **Depth of a Node:** The depth of a node is the number of edges from the root to a node p . The root node has a depth of 0.
- **Height of a Tree:** The height of a tree is the number of edges on the longest path from the root to any leaf. It represents the maximum depth of the tree.

Implementation

Trees can be implemented using two common approaches: a **linked structure** or an **array-based structure**.

In a **linked structure implementation**, the tree is composed of node objects. Each node stores a data element and references (links) to its children. This approach is flexible and naturally represents hierarchical relationships, making it suitable for general trees and binary search trees. However, it requires extra memory for storing references and may have less cache-friendly memory access.

```

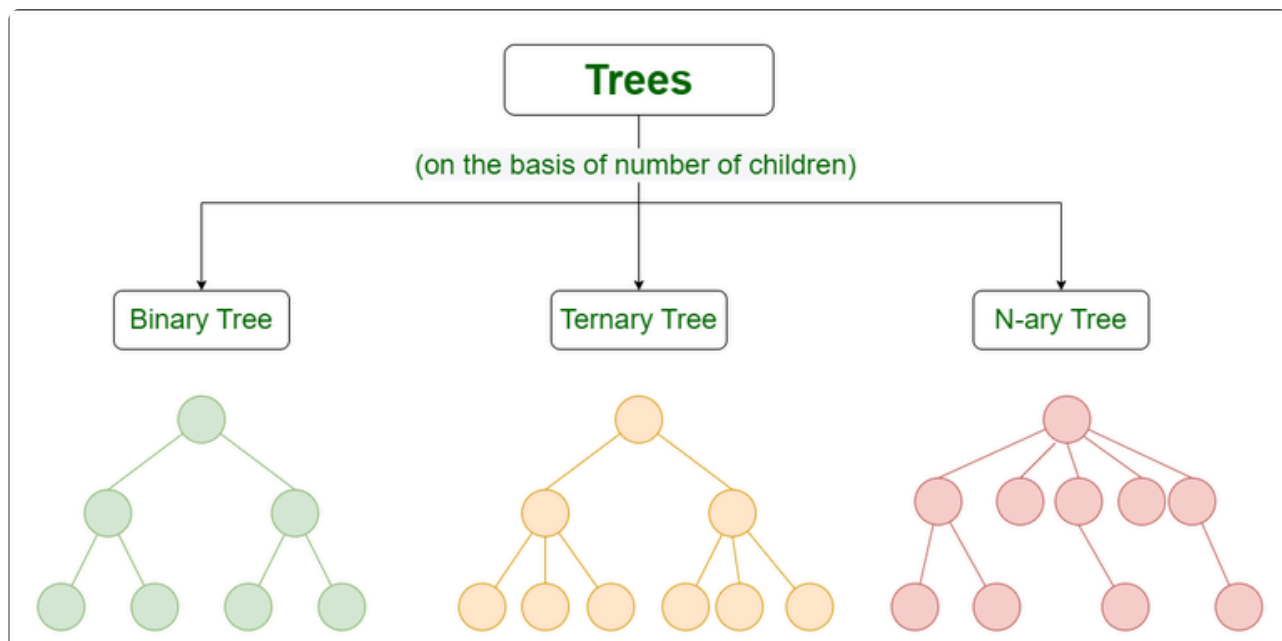
1  class Node<E> {
2      E element;
3      Node<E> left;
4      Node<E> right;
5
6      Node(E element, Node<E> left, Node<E> right){
7          this.element = element;
8          this.left = left;
9          this.right = right;
10     }
11
12     Node(E element) {
13         this(element, null, null);
14     }
15 }

```

In an **array-based implementation**, tree nodes are stored in an array, where parent-child relationships are determined using index calculations. This approach is simple and memory-efficient for **complete binary trees**, such as heaps. However, **it is less flexible**, as it may waste space for sparse trees and is not well-suited for dynamic or unbalanced tree structures.

In practice, linked implementations are preferred for flexible and dynamic trees, while array-based implementations are ideal for trees with a fixed and predictable structure.

Types of Trees



Trees have 3 types based on the maximum number of children each node could have:

1. **Binary Trees:** A tree where each node has **at most 2 children**, where a child node is labeled as being either a **left child** or a **right child**. The left child always precedes

the right child in the order of children. Binary trees are the foundation for many important structures, such as binary search trees, heaps, and expression trees.

2. **Ternary Trees:** A tree in which each node has at most **three children**, typically labeled as the **left**, **middle**, and **right** child. Ternary trees are less common than binary trees but can be useful when you want to represent data with three-way decisions or categories.
3. **N-ary Trees (or General Trees):** A tree in which each node can have **at most N children**, where N is any positive integer. There is no restriction on the number of children, and the children can be ordered or unordered depending on the application. N-ary trees are widely used in representing hierarchical data such as file systems, organizational structures, and XML/HTML documents.

More on Binary Trees

Since binary trees are amongst the most widely used tree data structures, they deserve a more detailed discussion. Binary trees can be classified based on different factors:

Binary Tree Types

Binary trees can be classified based on several criteria, including:

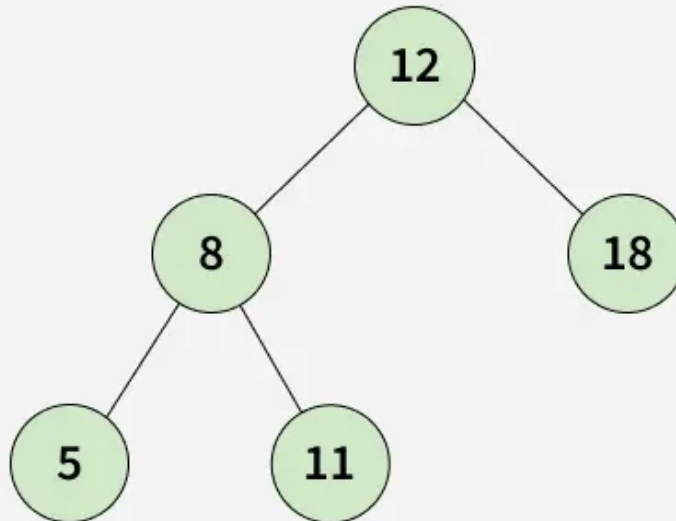
- number of children per node
- Structure or shape

Types based on the number of Children

1. Full Binary Tree

A binary tree is considered to be full if every internal node has exactly 0 or 2 children

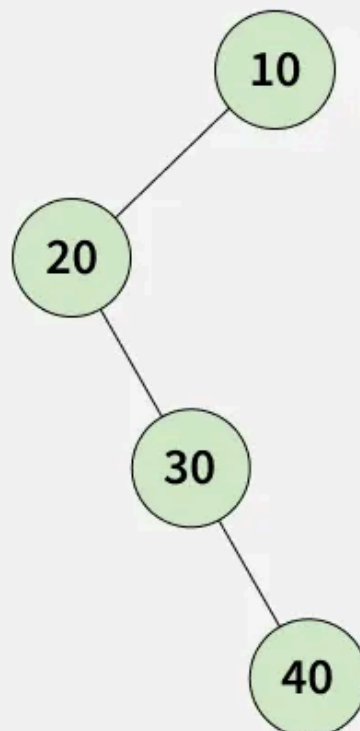
Full Binary Tree



2. Degenerate Tree

A tree is considered degenerate (or pathological) if each internal node has exactly one child. In this case, the tree effectively degenerates into a structure similar to a linked list. As a result, the time complexity of basic operations such as searching, insertion, and deletion becomes linear, which is comparable to that of a linked list.

Degenerate Tree



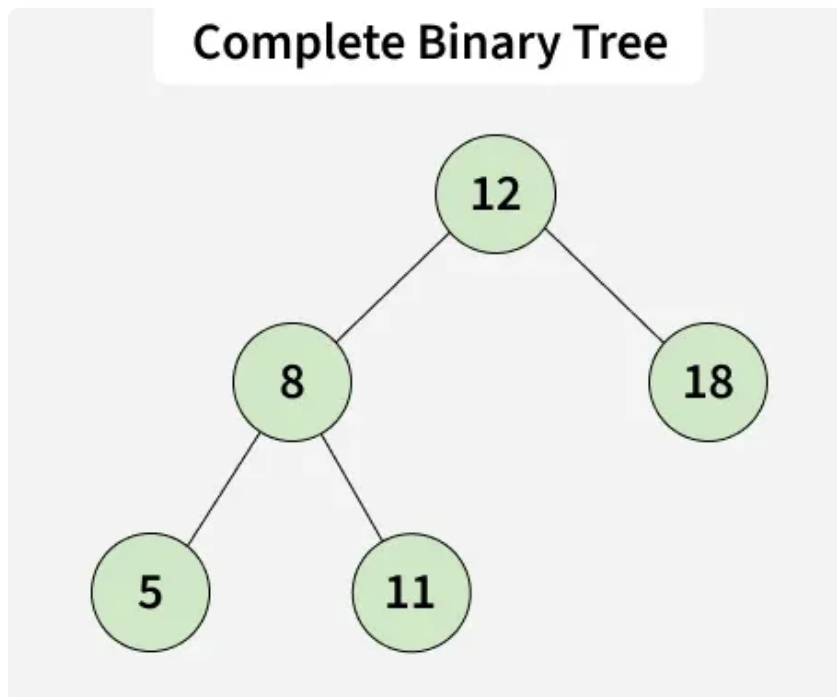
There is a special case from degenerate trees called a **skewed binary tree**, where all internal nodes have exactly one child, and the child is consistently either on the **left** (left-

skewed) or the **right** (right-skewed).

Types Based On Structure

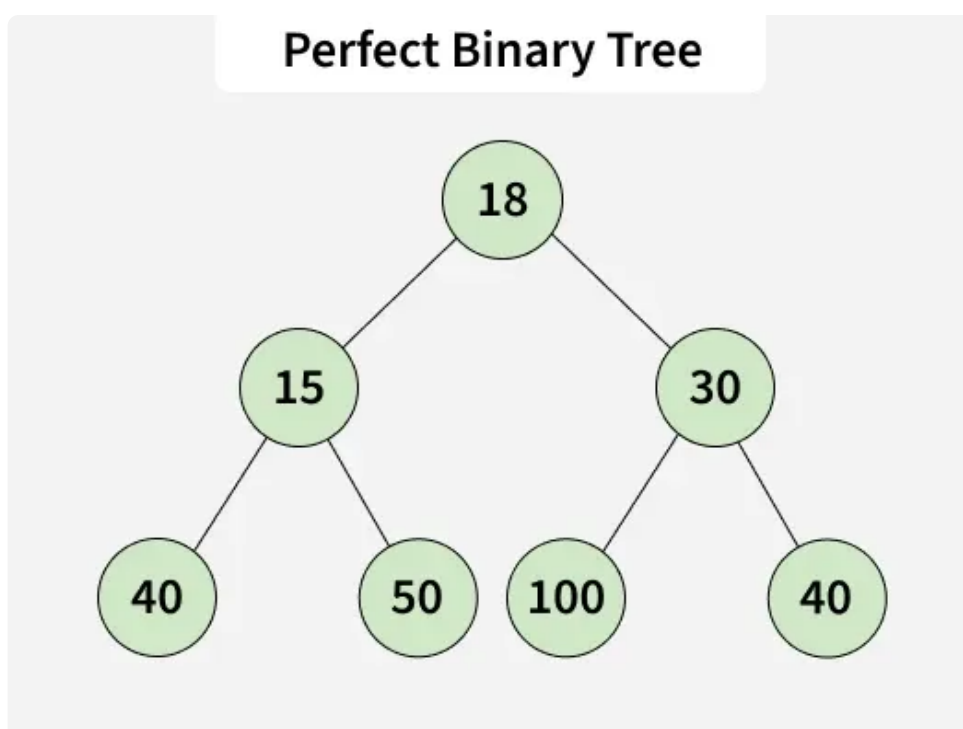
1. Complete Binary Tree

A binary tree is considered to be complete if all the levels are completely filled except possibly the last level, and the last level is filled from left side



2. Perfect Binary Tree

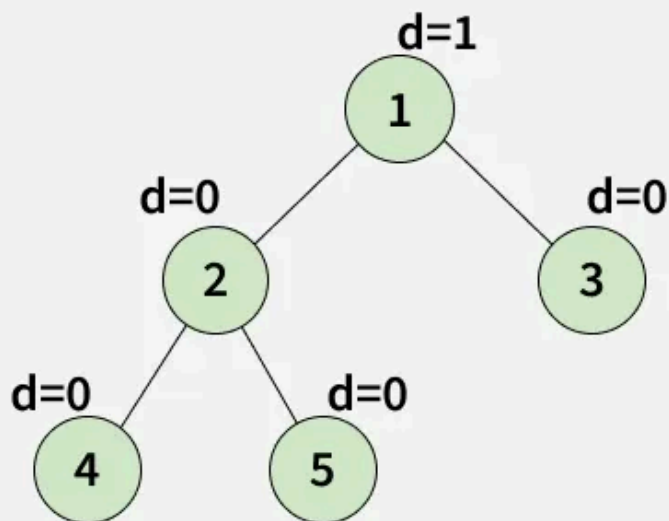
A binary tree is considered to be perfect if all internal nodes have exactly 2 children and all leaf nodes are at the same level



3. Balanced Binary Tree

A binary tree is considered balanced if for every node, the height difference between its left and right subtrees is at most one, ensuring the tree remains compact and operations like search, insert, and delete stay efficient $O(\log n)$

Balanced Binary Tree



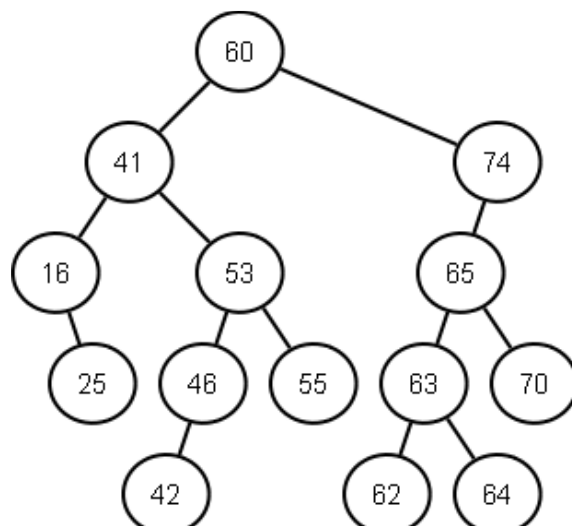
Depth of a node = $|\text{height of left child} - \text{height of right child}|$

Special Types of Binary Trees

1. Binary Search Tree (BST)

A binary search tree is a binary tree that has the following properties:

- The left subtree of a node contains only nodes with values less than the node's value
- The right subtree of a node contains only nodes with values greater than the node's value



This order allows us to visit the nodes in **ascending order** of their values using an **in-order traversal**

Binary Search Trees (BSTs) are useful for maintaining a **dynamically changing set of elements in sorted order**, allowing efficient search, insertion, and deletion operations. Unlike a sorted array, which requires shifting elements for insertions and deletions, a BST can handle these operations in **$O(\log n)$** time (on average) while still allowing an **in-order traversal** to visit elements in sorted order.

Take a look at this simple yet effective implementation of the binary search tree:

```
1  class Node {
2      int value;
3      Node left, right;
4
5      public Node(int value) {
6          this.value = value;
7          left = right = null;
8      }
9  }
10
11 class BST {
12     Node root;
13
14     // Insert a value into the BST
15     public Node insert(Node root, int value) {
16         if (root == null) {
17             return new Node(value); // Create a new node if tree is empty
18         }
19
20         if (value < root.value) {
21             root.left = insert(root.left, value); // Insert in left subtree
22         } else if (value > root.value) {
23             root.right = insert(root.right, value); // Insert in right subtree
24         }
25         // If value == root.value, do nothing (no duplicates)
26         return root;
27     }
28
29     public void inorder(Node root) {
30         if (root != null) {
31             inorder(root.left);
32             System.out.print(root.value + " ");
33             inorder(root.right);
34         }
35     }
36
37     public static void main(String[] args) {
38         BST tree = new BST();
39         int[] values = {50, 30, 70, 20, 40, 60, 80};
40
41         for (int v : values) {
42             tree.root = tree.insert(tree.root, v);
43         }
44
45         System.out.println("BST elements in sorted order:");
46         tree.inorder(tree.root);
47     }
48 }
```

2. AVL Tree

An AVL tree is a self-balancing binary search tree where the height difference between its left and right subtrees is at most one. This property ensures that the tree remains balanced, preventing it from becoming skewed. As a result, search, insertion, and deletion operations can be performed in **$O(\log n)$** time.

3. Red-Black Tree:

A red-black tree is a self-balancing binary search tree in which each node is assigned a color (red or black) and must satisfy specific coloring rules. These rules ensure that the tree remains approximately balanced, guaranteeing $O(\log n)$ time complexity for search, insertion, and deletion operations.

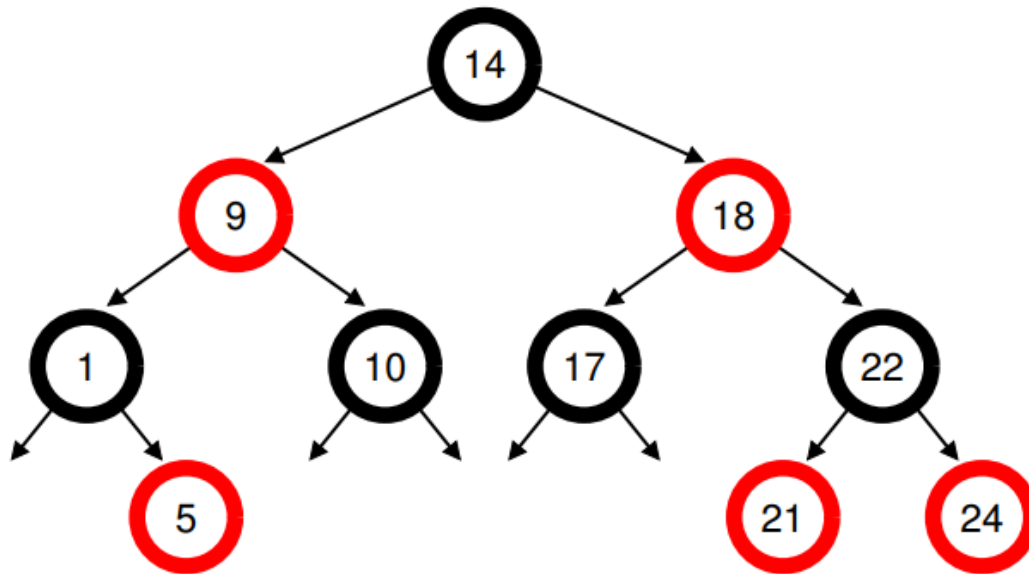


Figure 9.5: CFS Red-Black Tree

Red-black trees are used in the Linux kernel, most notably in the **Completely Fair Scheduler (CFS)**, to efficiently manage and schedule runnable processes.

Tree Traversal Algorithms

Tree traversal algorithms define the order in which the nodes of a tree are visited. Traversals are essential for accessing, searching, and processing tree data. Different traversal strategies are used depending on the problem being solved.

Pre-order

In pre-order traversal, the current node is visited first, followed by the left subtree, and then the right subtree. This traversal is commonly used to copy or serialize a tree.

Order: Parent -> Left -> Right

```
void preorder(Node root) {
    if (root == null) return;
    System.out.print(root.value + " ");
    preorder(root.left);
    preorder(root.right);
}
```

Post-order

In post-order traversal, the left subtree is visited first, then the right subtree, and finally the current node. This traversal is useful when deleting a tree or evaluating expression trees.

Order: Left -> Right -> Root

```
1 void postorder(Node root) {  
2     if (root == null) return;  
3     postorder(root.left);  
4     postorder(root.right);  
5     System.out.print(root.value + " ");  
6 }
```

In-order

In in-order traversal, the left subtree is visited first, followed by the current node, and then the right subtree. When applied to a Binary Search Tree, this traversal visits the nodes in ascending order.

Order: Left -> Root -> Right

```
1 void inorder(Node root) {  
2     if (root == null) return;  
3     inorder(root.left);  
4     System.out.print(root.value + " ");  
5     inorder(root.right);  
6 }
```

Breadth-First

Breadth-first traversal, also known as level-order traversal, visits the nodes level by level from left to right. It is typically implemented using a queue.

```
1 void breadthFirst(Node root) {  
2     if (root == null) return;  
3  
4     Queue<Node> queue = new LinkedList<>();  
5     queue.add(root);  
6  
7     while (!queue.isEmpty()) {  
8         Node current = queue.poll();  
9         System.out.print(current.value + " ");  
10  
11         if (current.left != null) queue.add(current.left);  
12         if (current.right != null) queue.add(current.right);  
13     }  
14 }
```

Real World Applications

Trees are widely used in real-world computing systems because they efficiently represent hierarchical data and support fast searching and organization.

Common applications of trees include:

- **File systems**, where directories and files are organized in a hierarchical tree structure.
- **Databases**, which use tree-based indexes (such as B-trees and B+ trees) to enable fast data retrieval.
- **Decision-making systems**, such as decision trees used in machine learning and game AI.
- **Compilers and interpreters**, where syntax trees and expression trees are used to analyze and evaluate code.

Tasks

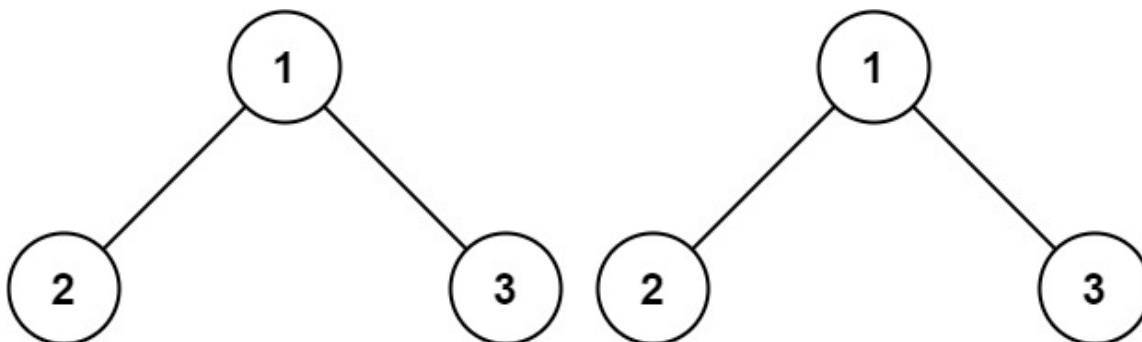
Task 1 (4 points)

Given the roots of two binary trees p and q , write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

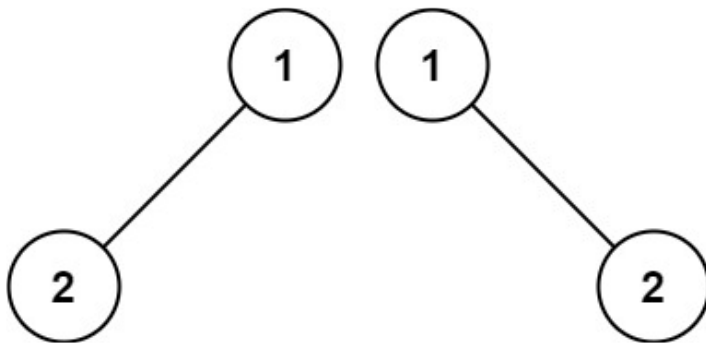
Question link on [LeetCode](#)

Example 1:



Input: $p = [1,2,3]$, $q = [1,2,3]$

Output: `true`

Example 2:

Input: $p = [1, 2]$, $q = [1, \text{null}, 2]$

Output: `false`

Constraints:

- The number of nodes in both trees is in the range $[0, 100]$.
- $-104 \leq \text{Node.val} \leq 104$

Task 2 (6 points)

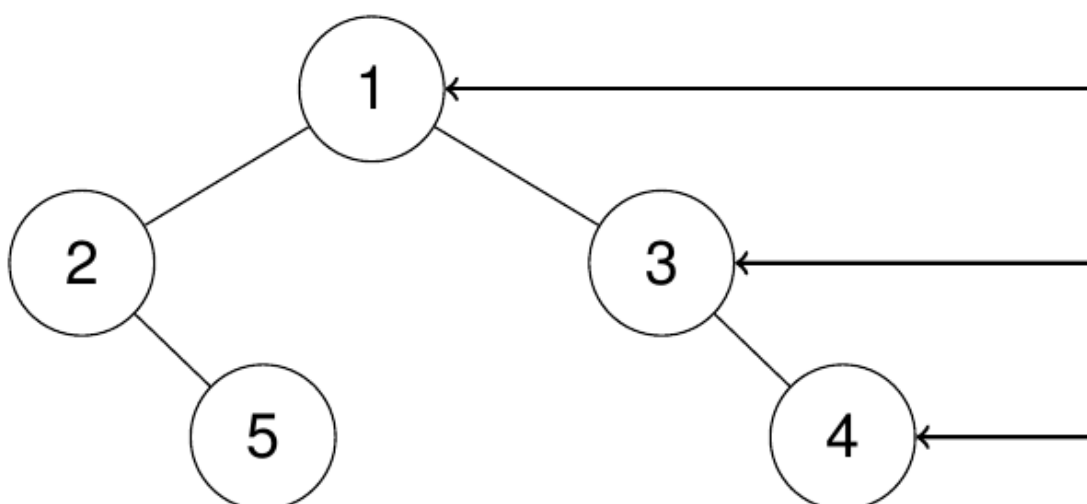
Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

Question link on [LeetCode](#)

Example 1:

Input: $\text{root} = [1, 2, 3, \text{null}, 5, \text{null}, 4]$

Output: $[1, 3, 4]$

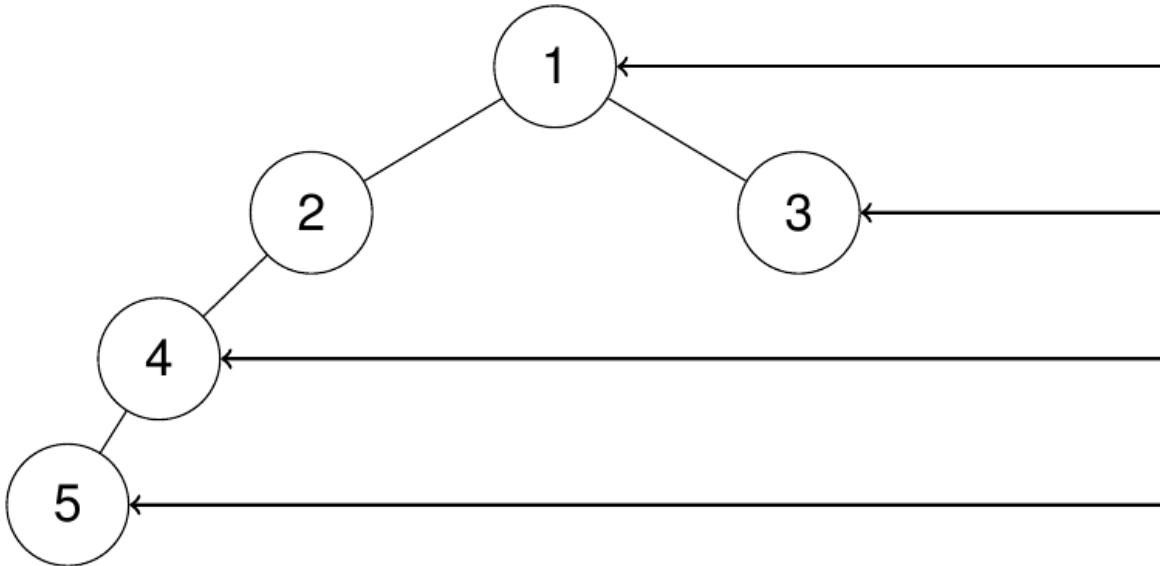
Explanation:

Example 2:

Input: root = [1,2,3,4,null,null,null,5]

Output: [1,3,4,5]

Explanation:



Example 4:

Input: root = []

Output: []

Constraints:

- The number of nodes in the tree is in the range $[0, 100]$.
- $-100 \leq \text{Node.val} \leq 100$

Tasks Submission Guide

You will solve both problems on LeetCode and you have to make sure that your solution passes all test cases and that LeetCode accepts it. Once you are done, its time to submit your solution on GradeScope.

Firstly, make sure to submit **only** two files:

1. TaskOne.java

2. TaskTwo.java

Secondly, for each question, you have to follow these steps to make sure GradeScope's autograder runs without issues:

For Task 1:

1. Copy your whole solution into a file named `TaskOne.java`
2. Change the class name from `Solution` to `TaskOne`, and make sure that the class is `public`
3. add the following package identifier to the beginning of the file. Make sure to copy it and **don't write it manually to avoid typos**:

```
1 | package com.gradescope.labSix;
```

Here's an example solution for reference:

```
1 | package com.gradescope.labSix;
2 |
3 | public class TaskOne {
4 |     public boolean isSameTree(TreeNode p, TreeNode q) {
5 |
6 |     }
7 |     /* if any helper methods are needed, make sure to include them */
8 | }
```

For Task 2:

1. Copy your whole solution into a file named `TaskTwo.java`
2. Change the class name from `Solution` to `TaskTwo`, and make sure that the class is `public`
3. add the following package identifier to the beginning of the file. Make sure to copy it and **don't write it manually to avoid typos**:

```
1 | package com.gradescope.labSix;
```

Here's an example solution for reference:

```
1 | package com.gradescope.labSix;
2 |
3 | public class TaskTwo {
4 |     public List<Integer> rightSideView(TreeNode root) {
5 |
6 |     }
7 |     /* if any helper methods are needed, make sure to include them */
8 | }
```

Happy Coding 😊

Last changed by

Ahmad Albarasy · Follow

0 5

Add a comment

Read more

Data Structures and Algorithms Lab 5 - Stacks and Queues

In this lab, we are going about stacks and queues, which are

Dec 13, 2025

Data Structures and Algorithms Lab 4 - Recursion

In this lab, we are going to learn about recursion and how can we use it to solve interesting problems. We ar...

Dec 6, 2025

Data Structures and Algorithms Lab 3 - Algorithm Analysis

In this lab, we are going to learn how to analyze our algorithms in terms of time and space complexity, whi...

Dec 3, 2025

Data Structures and Algorithms Lab 2 - Linked Lists

In this lab, we are going to learn about Linked Lists. What they are, how can we implement them, their key...

Dec 3, 2025

Read more from Ahmad Albarasy

Published on  HackMD